Regular Expressions

Ling 250/450: Data Science for Linguistics
C.M. Downey
Spring 2025

Regular Expressions

- Regular Expressions (aka “RE”, “RegEx”) are a powerful notation for **matching patterns in text**
- Most programming languages have their **own implementation** of regex
  - Python: the **re package** is included automatically
  - Can be accessed with `import re`
  - A version is also available in the **bash shell**
- Can capture a **huge variety** of patterns (though **not** all syntactic structure)
- Later: can be used to **find and replace** certain patterns in text

Basics

- Regular Expressions (REs) are encoded with **strings**
  - The **SLP book** uses `/slashes/` to denote REs. In class I’ll use `"quotes"`, since this is how **Python** denotes them
- REs can stand for **literal, case-sensitive strings**
  - `"woodchuck"` matches all occurrences of that string, including the sub-string in the word "woodchucks"
  - It does **not match** "Woodchuck", since RE is **case-sensitive**
  - **Note:** the RE `"the"` will match both the word "the" and sub-words like "other"

Sets

- Braces [ ] can be used to indicate **sets of characters**. It will match **any character within the braces**
  - Ex: `"[Ww]oodchuck"` matches both "Woodchuck" and "woodchuck"
  - It does **not match** "Wwoodchuck". The characters in the braces are **options**
- Braces can also contain a **range of characters**
  - Works for characters that have a **natural ordering**, e.g. `[a-z]`, `[A-Z]`, `[0-9]`
  - Can specify a **sub-range** like `[2-5]` (digits from 2 to 5)
- A back-slash can be used to indicate the **literal brace character**, e.g. `\[` and `\]`
  - This is known as **"escaping"** a character that otherwise has a special meaning

Counters

- Several operators are used to indicate **counts** of characters or patterns
- `"_?"` : **zero or one** of the preceding pattern
  - Ex: `"colou?r"` matches either "color" or "colour"
- `"_+"` : **one or more** of the preceding pattern
  - Ex: `"ba+"` matches "ba", "baa", "baaa", etc.
  - Sometimes called the **Kleene plus**
- `"_*"` : **zero or more** of the preceding pattern
  - Ex: `"ab*a"` matches "aa", "aba", "abba", "abbba", etc.
  - Sometimes called a **Kleene star**

Anchors

- Anchors refer to the **position** within a string
- ^ indicates the **beginning of the line** and $ indicates the **end of the line**
- Examples:
  - `"^Cat"` : the word "Cat" if it occurs at the beginning of a line
  - `"dog$"` : the word "dog" if it occurs at the end of the line
  - `"^The Cat$"` : the string "The Cat", if it is the **only content** of the line
- `\b` indicates a **word boundary** ("words" are strings of letters, digits, and underscores **without spaces**)
  - Ex: `"\bthe\b"` matches "pet the cat" but **not** "other"

Disjunction

- The **"pipe"** character (`|`) is used to indicate **"either/or"** (disjunction)
  - Ex: `"cat|dog"` matches either "cat" or "dog"
- RegEx has an **order of operations**. The disjunction **applies last**
  - So the previous example does **not match** "cadog" or "catog"
- In **some** places (like the SLP book), **parentheses** can be used to specify what the disjunction applies to (similar to parentheses in math)
  - Ex: `"gupp(y|ies)"` to match "guppy" or "guppies"
  - **Warning:** this is **not how RegEx behaves in Python!** (See next slide)

Parentheses

- In the SLP book, parentheses simply **indicate precedence** (e.g. which operations should be done first)
  - Ex: `"gupp(y|ies)"` matches "guppy" and "guppies"
- In **Python**, this behavior requires a `?:` added to the **opening parenthesis**
  - Ex: `"gupp(?:y|ies)"` matches "guppy" and "guppies"
- In **Python**, **regular parentheses** match **only** the part in parentheses
  - Ex: `"gupp(y|ies)"` matches "y" in "guppy" and "ies" in "guppies"
  - But **NOT** "y" in "puppy" or "ies" in "puppies"

Other basics

- Sets can be **combined** with counters. Ex: `"[a-z]+"` matches **one or more lowercase letter**
- At the **beginning of a set**, a **caret** character (^) means **"not"**
  - Ex: `"[^0-9]"` matches anything **except** digits
- The **period** character is the **"wildcard"**, which matches **any single character** (except the new-line character)
  - `"beg.n"` matches "begin", "begun", "began", "beg9n", etc.
  - A **literal period** can be indicated with a **slash**, ex: `"Stop\."`

Aliases

- Aliases are special sequences that **stand in for sets of characters**
- `\d` : any **digit** (equal to `[0-9]`)
- `\D` : any **non-digit** (equal to `[^0-9]`)
- `\w` : any **alphanumeric** character `[a-zA-Z0-9_]`
- `\W` : any **non-alphanumeric** character `[^a-zA-Z0-9_]`
- `\s` : any **whitespace** character (space, tab, newline)
- `\S` : any **non-whitespace** character

Advanced counters

- `_{n}` : **exactly n occurrences** of the previous pattern
- `_{n,m}` : **between n and m** occurrences of the previous pattern
- `_{n,}` : **at least n** occurrences of the previous pattern
- `_{,m}` : **up to m** occurrences of the previous pattern
- Examples:
  - `"(?:ba){5}"` → "bababababa"
  - `"(?:ba){3,5}"` → "bababa", "babababa", "bababababa"

Python RegEx functions

- `re.findall(pattern, string)`: find and return **all instances** of the RegEx pattern within the input string (returns a list)
- `re.search(pattern, string)`: search for the **first instance** of the RegEx pattern within the input string. Returns a `Match` object with information about the match (such as position within the string)
  - This function might be **more confusing** than `findall` when first starting out
- `re.sub(pattern, replacement, string)`: **returns a copy** of the input string with **all instances** of the pattern **replaced** with `replacement`

RegEx Tips

- **No need to memorize!** I almost always have a Regular Expressions "cheat sheet" open when I'm working with them
- Getting a RegEx right is about minimizing **false positives** and **false negatives**
  - False positives: strings that **match but should not**
  - False negatives: strings that **don't match but should match**
- The best way to learn is to **practice on real examples**
  - e.g. use the Python interpreter and test out your patterns on real text